A Combinatorial Test Suite Generator for Gray-Box Testing

Anthony Barrett, Daniel Dvorak
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA, USA
{firstname.lastname}@jpl.nasa.gov

***Abstract*—In black-box testing, the system being tested is typically characterized as a number of inputs, where each input can take one of a number of values. Thus each test is a vector of input settings, and the set of possible tests is an N dimensional space, where N is the number of inputs. For example, an instance of a simulation of a Crew Exploration Vehicle's (CEV) launch pad abort scenario can have 76 floating-point inputs. Unfortunately, for such a large number of inputs only a small percentage of the test space can be actually tested. This paper characterizes levels of partial test space coverage and presents Testgen, a tool for generating a suite of tests that guarantees a level of test space coverage, which a user can adapt to take advantage of knowledge of system internals. This ability to adapt coverage makes Testgen a gray-box testing tool.**

Keywords-software testing; combinatorial testing; gray-box testing

I. INTRODUCTION

Typically testing is a black art where a tester poses a suite of problems that exercises a system's key functionalities and then certifies correctness once the system passes those tests. These problems can vary from using a small number of hand made tests that check if a system catches invalid inputs and responds appropriately given small off-nominal perturbations to using all possible tests that check the responses to all possible inputs and perturbations. While testing the response to small off-nominal perturbations is often done, exhaustive testing is rarely done due to a combinatorial explosion with the number of inputs and perturbations. Instead, testing takes the form of a randomly generated set of inputs and perturbations to sample a targeted area in the space of possibilities. Unfortunately this tactic gets ever more problematic as systems and software get larger and more complicated, resulting in using huge Monte Carlo test suites to get an informal level of confidence.

While there are numerous approaches toward testing, each approach falls into one of three classes depending on how much information a test engineer is provided during test suite generation. The simplest is *black box* testing, where a test engineer is just given the inputs and what values they can take. On the opposite end of the spectrum, *white box* testing gives access to the system's internals for inspection. Between these two extremes, *gray box* testing gives partial information on a system's internals, to focus testing.

This paper discusses a combinatorial alternative to random testing and how to extend it to gray box testing. For instance combinatorial techniques enable exercising all interactions between pairs of twenty ten-valued inputs with only 212 tests. More precisely, any two values for any two parameters would appear in at least one of the 212 tests. While this number of tests is miniscule compared to 10^{20} possible exhaustive tests, anecdotal evidence suggests that they are enough to catch most coding errors. The underlying premise behind the combinatorial approach can be captured in the following four statements, where a factor is an input, single value perturbation, configuration, etc.

- The simplest programming errors are exposed by setting the value of a single factor.
- The next simplest are triggered by two interacting factors.
- Progressively more obscure bugs involve interactions between more factors.
- Exhaustive testing involves trying all combinations of all factors, but is usually intractable.

So errors can be grouped into families depending on how many factors need specific settings to exercise the error. The *m*-factor combinatorial approach guarantees that all errors involving the specific setting of *m* or fewer factors will be exercised by at least one test.

To generate 2-factor (or pairwise) combinatorial test suites there are a number of algorithms in the literature [1], and our algorithm is a generalization of the In-Parameter-Order pairwise test suite generation algorithm [2], which facilitates gray-box testing by including test engineer desired capabilities to:

- explicitly include particular seed combinations,
- explicitly exclude particular combinations,
- require different *m*-factor combinatorial coverage of specific subsets of factors, and
- nest factors by tying the applicability of one factor to the setting of another.

With the exception of nested factors, these capabilities have been discussed in the literature. PICT [3] adds all but the last capability to a combinatorial test-suite generator, while Cohen, Dwyer, and Shi [4] describe a generic way to add combination exclusions to a number of combinatorial test-suite generators using a Boolean SAT solver. The contribution here involves adding these capabilities to an In-

Parameter-Order algorithm, resulting in a system that is fast enough to handle problems with up to a thousand factors.

The next two sections of this paper subsequently explain combinatorial testing and how it can provide test space coverage guarantees, and discusses the new features desired by a test engineer. Given these extra features, the following sections present a generalized version of the IPO algorithm, which provides such a guarantee; describe experiments and applications of a JAVA implementation, which is competitive with other pairwise algorithms while also scaling to real world problems; and conclude by discussing future work.

II. Coverage via Combinatorial Testing

From a geometric perspective, system testing is a matter of exploring a K-dimensional test space in search of K factors that cause the system to exhibit an error. Given some way to evaluate a particular test, the main problem that a tester faces is the selection of which tests to perform. Since each test takes time to perform, there is a strong desire to minimize the number of tests. On the other hand, there has to be enough tests to exercise the system as well as needed.

A. Pairwise vs Random Testing

The most commonly used form of combinatorial testing is pairwise testing. Instead of all possible combinations of all test factors (exhaustive testing), a generated test suite covers all possible combinations among pairs of test factors. For instance, testing a system having three binary test factors (such as three switches named A, B, C) with exhaustive testing (all possible combinations) requires eight tests. However, if a test engineer determines that it is adequate to just test all pairwise combinations among the three switches, then only four tests are needed, as shown in Table I (given any pair A-B, B-C, and A-C, all four combinations of values appear). While the savings here is only from eight to four, it rapidly increases with the number of test factors. For instance, given twenty 10-level factors, all pairwise interactions are testable with 212 or fewer tests, resulting in at least a 10^{20} to 212 reduction.

TABLE I. A four-element test suite that tests all pairwise interactions among three binary factors.

Test Factor:	A	B	C
Test 1:	0	0	0
Test 2:	0	1	1
Test 3:	1	0	1
Test 4:	1	1	0

The premise of pairwise test generation is that exercising interactions among factors with finitely enumerated levels will discover many of a system's defects, and interactions among factors can be exercised by testing all possible *combinations* of factor levels. Accordingly, pairwise testing involves generating test suites that exercise all combinations of levels for any possibly interacting pair of factors with as few tests as possible. In our 10^{20} example, the number of pairs of factors is $20 \times 19/2 = 190$, and the number of combinations for each pair of parameters is 10^2. Since each test will exercise 190 combinations, it is theoretically impossible to test all combinations with less than 100 tests.

While good in theory, generating minimal test suites is computationally intractable. Thus different algorithms for pairwise testing take heuristic approaches to generating test suites. While the number of tests generated is often quite small, there is no guarantee that it is minimal. For instance, in the 10^{20} example, the minimum number of pairwise tests must be more than 10^2, but it is less than the 212 – of all the heuristic test tools, the best result found so far is 180.

Unlike pairwise testing, random testing generates each test's parameters completely at random, making no attempt at minimization. Thus random testing is much simpler than combinatorial testing. Still, as shown in Figure 1, random testing performs quite well. For instance, given 212 randomly generated tests, there is only a 0.99^{212} probability (or 22% chance) that any particular pair of interacting parameters is not checked. While this result makes random testing look comparable with pairwise testing [5], an interest in the probability that all pairwise interactions are checked results in Figure 2's probability graph, showing that random testing takes around 7 times as many tests to achieve a approach a pairwise guarantee in this example. Thus pairwise testing is an improvement on random testing when a coverage guarantee is required in black-box testing.

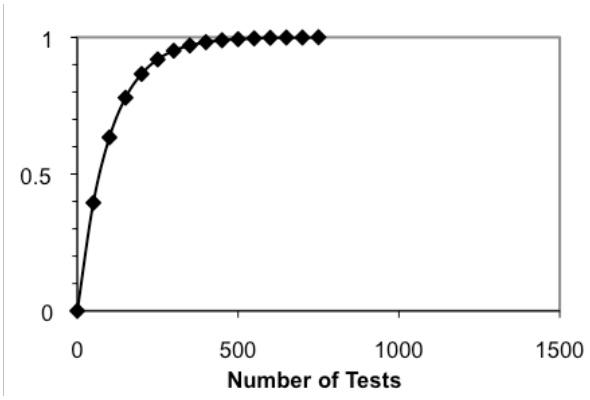

Figure 1. Probability that a randomly generated test suite will cover a particular pair of parameter assignments in a 10^{20} system.

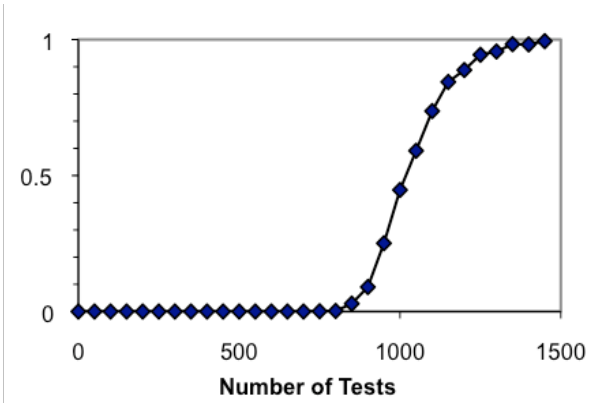

Figure 2. Probability that a randomly generated test suite will cover all pairwise parameter assignments in a 10^{20} system.

B. Gray-Box Combinatorial Testing

While the number of tests performed using the pairwise approach is miniscule compared to exhaustive testing and much smaller than random testing, anecdotal evidence has suggested that this small number of tests is enough to catch most coding errors [6,7,8,9], which provides some support for the underlying premise behind a combinatorial approach. Still, pairwise testing has all the limitations of a black box approach [5], and this paper focuses on adding capabilities to a combinatorial test suite generator to facilitate its use in gray box testing.

For instance, given a set of expected use cases, a test engineer can make sure that a specific set of tests are included in the generated test suite. Similarly, when told that specific combinations of factor assignments will never occur, a test engineer would wish to assure that the generated test suite would exclude such combinations. In addition to inclusions and exclusions, a test engineer would wish to generate stronger n-way combination tests for those subsets of factors that are highly interacting. Finally, if one factor's setting makes a system abort, a subsequent factor is never even testing. Thus when testing failure scenarios, it is often the case where one factor's mere existence depends on the setting of another, and a test engineer must take that into account when crafting a test suite.

III. Components Of A Test Model

Classically, a combinatorial test-suite generator's input is a set of factors, and its output is a set of test vectors, where each factor is defined as a finite set of levels and the i^{th} element of each test vector is an element of the set of levels T_i for the i^{th} factor. When generating a pairwise, or 2-way, test-suite the computed set of vectors (M) are such that for any 'a' in T_i and 'b' in T_j there is some vector *m* in M such that *m*[i] is 'a' and *m*[j] is 'b', and this definition extends to higher n-way test suites. For instance, in a 3-way test suite the vectors in M are such that for any 'a' in T_i, 'b' in T_j, and 'c' in T_k there is some vector *m* in M such that *m*[i] is 'a', *m*[j] is 'b', and *m*[k] is 'c'.

> Input: $[T_1 \ldots T_k]$ – k enumerated sets denoting the factors
> Output: M – a set of k-element test vectors.

This definition makes an assumption that factors have finite numbers of levels, but parameters can take floating-point values resulting in factors having infinite numbers of levels. A better approach to handling floating point numbers involves discretely partitioning floating-point ranges, generating a suite of tests that assign ranges to floating point factors, and then randomly selecting values from ranges when performing a test. This approach was taken when testing CEV simulations where all of the parameters were floating point ranges.

A. Nested Factors

A second assumption inherent in combinatorial testing involves the independence of factors. This assumption seriously limits the applicability of combinatorial testing. Many systems exhibit a property where the mere applicability of a parameter depends on the setting of another. For instance setting one parameter to an illegal value can result in a system halting with an error message, and interactions between other factors are eclipsed by this halt.

Nested factors addresses this limitation, and the set NEST for representing nested factors is defined as follows, where the level of previous factors determines the applicability of later factors. Thus NEST defines a hierarchy of factors where earlier factors control the applicability of later ones.

> $NEST \subseteq \{(N(i),c(i),i) \mid 1 \leq N(i) < i \leq k \text{ and } c(i) \in T_{N(i)}\}$, where N(i) and c(i) denote that the i^{th} factor applies only when the $N(i)^{th}$ factor is level c(i).

Using nested factors, a test engineer can do more than just handle the testing of error messages. Most programs exhibit a nested block structure of conditionals. Using nested factors, a test engineer can define a combinatorial test suite that conforms to the block structure within a program. Resulting in being able to take advantage of code inspection when producing a test suite.

B. Seed Test Cases

The most obvious step to adding an ability to control combinatorial test generation involves specifying tests to include from a defined number of use cases. Testgen generalizes on this by letting a test engineer partially define tests to include. As such, SEEDS are defined as follows, where a '*' in i^{th} position is a wildcard that can be any level from the set T_i for the i^{th} factor. Within this definition, a set conforms with NEST when a value in the i^{th} position of a test vector implies that c(i) is in the $N(i)^{th}$ position whenever (N(i),c(i),i) is in NEST.

> $SEEDS \subseteq (T_1 \cup \{\text{'*'}\}) \times \ldots \times (T_k \cup \{\text{'*'}\})$, conforming to NEST and denoting specific combinations that must occur in returned tests.

Thus a test engineer defines specific combinations of factor levels to include using SEEDS, and a combination becomes a complete test when it lacks wildcards. A simpler alternative approach to including specific test cases involves just appending them to a test suite, but that results in more tests than necessary since an appended k-factor test would result in needlessly testing k(k-1)/2 combinations twice when doing pairwise testing. Testgen adds the seeds to a test suite first and then adds extra tests as needed to generate the combinatorial test suite.

C. Excluded Combinations

Complimentary to requiring the inclusion of specific seed combinations, a test engineer also needs the ability to exclude specific combinations. Essentially, when certain combinations are known to be illegal, a test suite generator should not produce them. For this reason the set EXCLUDE is defined as follows, where excluded combinations can have any number of wild cards. The requirement is that no generated test can be produced from an excluded combination by replacing the wildcards.

EXCLUDE $\subseteq$ (T1 $\cup$ {'*'})×...×(Tk $\cup$ {'*'}), consistent with elements of SEEDS and denoting specific combinations that cannot occur in returned tests.

Given this definition, keeping SEEDS and EXCLUDE consistent is a matter of assuring that no element of SEEDS can force the inclusion of a test that is explicitly ruled out by an element of EXCLUDE. For instance, the following elements of SEEDS and EXCLUDE are incompatible due to the fact that any test forced by the seed is explicitly prohibited. Note how replacing wildcards in the example exclude can generate any test generated by replacing wildcards in the example seed.

[1 0 2 3 * * 2 * * 7 * * * * 3 * * * * *] $\in$ SEEDS
[* 0 2 * * * 2 * * * * * * * * * * * *] $\in$ EXCLUDE

D. *Mixed Strength Coverage*

While most combinatorial test-suite generators focus on generating test suites with pairwise coverage, it has been shown that there are times when higher *n*-way coverage is motivated [10]. Unfortunately, the number of tests generated tends to explode with increasing *n*, and even pairwise testing between some factors is unnecessary [11]. To limit this explosion, a test engineer needs the ability to focus where n-way interaction coverage is applied, and COMBOS provides this facility with the following definition.

COMBOS $\subseteq$ {(*n*:t_1 ... t_j) | *n* ≤ j and 1 ≤ t_1 < ... < t_j ≤ k} denoting the required *n*-way combinations for specific subsets of *n* or more factors.

Using this feature, a test engineer can specify test suites that test any n-way interaction of any subset of test factors. For instance, the following set contains three elements that specify a desire to test pairwise interactions across three factors, three-way interactions across three factors, and one-way interactions across the last four factors. As this example implies, arbitrary overlaps are possible as well as non-interacting factors. In the example, the first five factors do not interact with the last two, and the last two are only tested to make sure that each level appears at least once in a test.

{ (2:1 2 3), (3:2 3 4), (1:5 6) }

Essentially, each COMBOS entry corresponds to a set of patterns that must appear in the generated test suite. For instance, the first COMBOS element above denotes the following twelve patterns if the first three factors are binary in the test model.

[0 0 * * * *] [0 1 * * * *] [1 0 * * * *] [1 1 * * * *]
[0 * 0 * * *] [0 * 1 * * *] [1 * 0 * * *] [1 * 1 * * *]
[* 0 0 * * *] [* 0 1 * * *] [* 1 0 * * *] [* 1 1 * * *]

E. *Repeats and Randomness*

The final feature applied by the Testgen system involves injecting randomness. It turns out that there are times when a test engineer would want to generate a test suite with more than one independent test of every possible interaction. To provide this feature, randomness is injected into the algorithm at specific points. While the system is deterministic for a given random seed, changing that seed provides very different test suites whose size varies slightly.

IV. TESTGEN ALGORITHM

Extending on the IPO algorithm [12], Testgen builds a test suite by focusing on each factor in order of its position – from left to right. As shown in Figure 3, Testgen starts by initializing the elements of M, with SEEDS to assure that seed combinations will be included in the resultant test suite. As such, the earlier example of a seed shows that each m $\in$ M is a vector of k elements for the k factors, and each element m[i] can be either a factor level from T_i or the wildcard "*". The main extensions over IPO revolve around the initialization with SEED, how combinations are computed, and the effort needed to replace wildcards at the end.

Testgen([T_1…T_k], SEEDS, NEST, EXCLUDE, COMBOS)

1. M ← SEEDS.
2. For *i* ← 1 to k do:
3. π_i ← {combinations that end with T_i, conforming with COMBOS, NEST, and EXCLUDE};
4. If π_i is not empty then
5. Grow tests in M to cover elements of π_i
6. Add tests to M to cover leftover elements of π_i
7. For each test m $\in$ M do:
8. For *i* ← 1 to k do:
9. If m[*i*] = '*' then
10. Randomly set m[*i*] to a value from T_i (conforming with EXCLUDE and NEST).
11. Return the test suite M.

Figure 3. Testgen algorithm

As illustrated in the pseudo-code, steps 2 through 6 form the heart of the algorithm by iterating through each factor in order. As described in the previous section, each COMBOS entry defines a set of combinations, each of which must appear in some test if it was not specifically ruled out by either EXCLUDE or NEST. Each combination is essentially a pattern that must be merged into the growing set of test vectors by either replacing wildcards in M with actual levels or by adding tests to M. For instance if we have the following element m $\in$ M and pairwise combination P $\in \pi_3$, m can cover P by setting m[3] to 1$\in T_3$.

[1 0 *] = m $\in$ M
[1 * 1 * * * * * * * * * * * * * * * * * * *] = P $\in \pi_3$
[1 0 1 * * * * * * * * * * * * * * * * * * *] – m covering P

While the actual implementation uses a more efficient way to represent combinations, the algorithm is easier to explain in terms of k-element vectors, so that is the representation used here. For instance, at each computation of π_i the iterate *i* is used to specify that P[*i*] is not a wildcard and P[*j*] is a wildcard for all *j* > *i*. Thus *i* partitions each COMBO entry's associated set of combinations in order to address each factor in order. As such, computing π_i involves iterating over the COMBO entries to compute the set of

combinations for the i^{th} partition. For instance, in our previous example of the combinations associated with a COMBO entry, the first line is associated with π_2, and the combinations in the following two lines appear in π_3.

Making π_i conform to NEST involves replacing wildcards in P $\in \pi_i$ as required by NEST. For instance, suppose that (1,0,6) $\in$ NEST, which requires that P[1]=0 whenever P[6]≠"*". For instance, the three element COMBOS example combines with the above NEST element to make π_6 = { [0 * * * * 0], [0 * * * * 1] }, where the elements in the last position derive from (1:5 6) $\in$ COMBOS while the 0 in the first position are subsequently set to conform with NEST. Thus replacing wildcards as required by NEST refines combinations, and a combination is removed if either the NEST refinement tries to change a combination value that is not a wildcard or the resultant combination is ruled out by EXCLUDE.

After computing π_i, the set M is extended by steps 4 through 6 to cover all of π_i's combinations. When there are no entries in π_i, no changes to M are necessary. Otherwise M is extended both horizontally and vertically to cover the combinations in π_i using the algorithms in Figures 4 and 5 respectively.

After iterating through each factor, M will cover all interacting combinations that a test engineer is interested in checking, but some of the tests will still have wildcards. Lines 7 through 10 resolve this issue by randomly setting wildcards to actual values that conform with EXCLUDE and NEST. Essentially, a wildcard is left in a position if the NEST specification determines that a factor is not applicable to a particular test. Also, the randomly selected value is restricted to assure that replacing a wildcard does not produce a test that is explicitly ruled out by EXCLUDE.

Finally, there is a possibility where EXCLUDE will rule out all possible values for a wildcard. This happens when the EXCLUDE set is either inconsistent, or large and complex. In this event the test engineer is informed of the problem. This algorithm makes no attempt to handle such cases since they are NP-complete, which can be proved by reducing the problem to SAT. The mAETG_SAT algorithm [4] points to a way to change the algorithm to better handle this problem using a SAT solver, but that approach is problematic when dealing with thousands of factors.

A. *Growing Tests*

Replacing wildcards at the i^{th} position grows the tests in M from left to right to make them cover the combinations in the current partition π_i. Testgen's heuristic approach toward selecting elements to replace these wildcards is defined by the pseudo-code in Figure 4, which is a generalization of the horizontal extension algorithm IPO_H [12] to keep excluded patterns and nesting in mind when extending tests to the next factor. As such, it starts by taking each element of T_i, and finds some test m $\in$ M where m[i] either is that element or can be set to it. Since different elements of T_i appear in different subsets of π_i, this is a very quick way to cover a large number of elements in π_i. After step 3 removes all covered tests from π_i, steps 4 through 7 replace wildcards in the i^{th} position of each test in order to greedily cover as many combinations as possible.

To grow tests in M to cover elements of π_I

1. For each c $\in T_i$ in random order do:
2. Find m $\in$ M where m[i] $\in$ {'*', c} & let m[i] ← c (conforming with EXCLUDE and NEST).
3. Remove elements from π_i that are covered by tests.
4. For each test m $\in$ M if π_i not empty do:
5. If m[i] = '*' then
6. Set m[i] to a level covering the most elements of π_i (conforming with EXCLUDE);
7. Remove covered elements from π_i

Figure 4. Algorithm for growing tests horizontally

As it stands, only lines 2 and 6 replace wildcards in tests and they never perform a replacement that is explicitly not allowed by EXCLUDE or NEST. While line 2 needs to explicitly conform to NEST and EXCUDE, line 6 only needs to take EXCLUDE into account. It turns out that line 6 implicitly takes NEST into account since NEST was used to alter the members of π_i. Setting m[i] to a level that covers elements of π_i implies that the level already conforms to NEST.

B. *Adding Tests*

While growing tests to greedily cover elements of π_i does result in removing many combinations, there are often times when growing tests will not cover all combinations. For those uncovered combinations, as well as the case where M's initially being empty, the routine outlined in Figure 5 will add tests to M to cover each leftover combination P left in π_i. As such, the routine iterates over each combination and tries to first replace wildcards in some test in order to cover P. For instance, consider the following test and combination. The test can be extended to cover the combination by modifying the wildcards in m[1], m[3] and m[7]. Notice that unlike the vertical growth routine, the horizontal growth routine can replace wildcards that precede the i^{th} position. Also, the routine can only replace a wildcard if the result does not violate an exclusion requirement.

[* 0 * * 1 * * * * 6 * * 7 * * * * * * * * *] = m $\in$ M
[1 * 1 * * * 2 * * 6 * * * * * * * * * * * *] = P $\in \pi_{10}$
[1 0 1 * 1 * 2 * * 6 * * 7 * * * * * * * * *] – m covering P

Finally, the third line of the routine tacks P to the end of M as a new test if there is no way to alter an existing test to cover P. Thus this routine can add tests to M, and will when M is initially empty.

To add tests to M to cover leftover elements of π_i

1. For each P left in π_i do:
2. Try to set '*' entries of some m $\in$ M to cover P (avoiding EXCLUDE);
3. If P still uncovered add a new test to M for P.

Figure 5. Algorithm for growing tests vertically.

V. EXPERIMENTS

The resultant implementation is 1041 lines of documented java code, and even with its extra capabilities the algorithm generates test suites that are comparable to those generate by the more restricted systems in the literature. As shown in Table 2, the code generates solutions that are comparable to other pairwise test-suite generators. In the problem sizes, the X^Y syntax means that there are Y X-valued parameters. Unfortunately, there are no established benchmark problems for the more capable systems, and papers on systems do not report performance, only solution quality. Testgen solved all of these problems in less than a second, and solved 3^{1000} with 48 tests in 22 sec.

TABLE II. SIZES OF PAIRWISE TEST-SUITES GENERATED BY VARIOUS TOOLS FOR VARIOUS PROBLEMS.

Problem	IPO[12]	AETG[13]	PICT[3]	Testgen
3^4	9	11	9	9
3^{13}	17	17	18	19
$4^{15}3^{17}2^{29}$	34	35	37	35
$4^{1}3^{39}2^{35}$	26	25	27	29
2^{100}	15	12	15	15
10^{20}	212	193	210	212

A. Related Work

The two efforts most related to this work involve extending the IPO algorithm from pairwise to user specified n-way combinatorial test suite generation with a system called IPOG [10], and work on extending the AETG [13] pairwise test generator to let a user specify numerous enhancements similar to ours in a system called PICT [3]. While IPOG is an extension of the IPO algorithm, its focus is solely on generalizing the algorithm from pairwise testing to n-way testing. Thus the result is still inherently focused on black-box testing where a test engineer can only specify the strength of the test.

On the other hand, PICT does extend a pairwise test generation algorithm to have many of the capabilities that Testgen provides. The main differences are the underlying algorithms and different capabilities provided. While Testgen is an extension of the $O(d^3n^2\log(n))$ IPO algorithm, PICT is based on the $O(d^4n^2\log(n))$ AETG algorithm [13].

The main feature that Testgen provides that PICT does not involves the ability to nest parameters, which facilitates making parameters depend on each other. In PICT each parameter is still independent with the following exception, PICT allows the definition of negative values such that a test can only have one negative value appear in a given test. This feature was motivated by failure testing just like nesting, but it is more restricted than nesting in that it only applies to failure testing.

Wang, Nie, and Xu [11] experiment with extending both an IPO based algorithm and an AETG based algorithm to replace simple pairwise test generation with generating test suites for interaction relationships. These relationships are specializations of COMBOS entries where n in $(n{:}t_1 \ldots t_j)$ is always equal to the number of t_i entries. Thus Testgen's test model specification language subsumes specifying interaction relationships.

Finally, Cohen, Dwyer, and Shi [4] present a general way to extend an algorithm to add EXCLUDE entries through the use of a SAT solver. By virtue of using a SAT solver, this approach is more apt to avoid the occasional case where Testgen leaves a wildcard in a test vector, but this is at the computational cost of continually calling a SAT solver whenever replacing a wild card.

B. Application to ANTARES Simulations

While Testgen has a general standalone utility, its primary use has been in an analysis feedback loop connected to two different Advanced NASA Technology Architecture for Exploration Studies (ANTARES) simulations of re-entry guidance algorithms [14] with 24 to 61 floating point setup parameters and the Crew Exploration Vehicle Launch Abort System [15] with 84 floating point setup parameters. To handle these floating-point factors, an analyst specifies ranges of interest and the granularity with which to partition each range. Given these partitions, Testgen can generate tests using factors with finite numbers of levels.

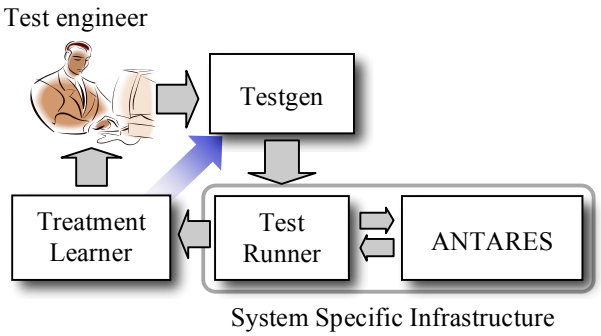

Figure 6. A feedback loop for analyzing ANTARES simulations

As illustrated in Figure 6, a test engineer generates a test model with the initial test space coverage requirements. This model is used by Testgen to define an initial set of test simulations. After the simulations are analyzed to classify there respective tests, the classified test vectors are passed to a treatment learner [16], which determines conjuncts of setup parameter ranges that drive the simulation to undesirable outcomes. These conjuncts both give a test engineer an improved comprehension of the results as well as motivate changes to the test model for more focused coverage around problem areas.

VI. CONCLUSIONS

This paper presents Testgen, a combinatorial test suite generator that can be used for gray-box testing. By giving a test engineer a large degree of control over what test space coverage guarantees a generated test suite provides, Testgen facilitates tuning tests in response to analyzing a system's internals. In addition to handling manually tuned testing requirements, Testgen has also been folded into a testing feedback loop where initial coverage requirements are further refined to explore the regions in a high-dimensional test space where a tested ANTARES simulation exhibits undesirable (or desirable) behaviors. The main advantage of

Testgen over Monte Carlo approaches when dealing with these simulations derives from improving coverage of the test space in less time.

While initial results are quite promising, there are several directions for further improvement both within Testgen and with how Testgen is used in an automated analysis feedback loop. While Testgen's speed enables generating test suites for systems with over a thousand parameters, improved speeds are possible by applying tricks to reuse old results when computing the next set of combinations π_i as i increases. With respect to use in a feedback loop, Testgen and a treatment learner are loosely coupled, where a full test suite is computed, simulation/learning occurs, and then another full test suite is computed. Another direction for improvement involves more tightly coupling the loop to make Testgen immediately alter a test suite upon learning a region of interest. Finally, the test suite specification is quite rich, which facilitates using static program analysis for generating tests.

ACKNOWLEDGEMENT

This work was performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The author would also like to thank Karen Gundy-Burlet, Johann Schumann, and Tim Menzies for discussions contributing to this effort.

REFERENCES

[1] M. Grindal, J. Offutt, and S. F. Andler, "Combination Testing Strategies – A Survey." Software Testing, Verification and Reliability, 15(3):167-199. 2005.

[2] Y. Lei and K. C. Tai, "In-Parameter-Order: A Test Generation Strategy for Pairwise Testing." Proc. Third International High-Assurance Systems Engineering Symposium, 1998.

[3] J. Czerwonka, "Pairwise Testing in Real World: Practical Extensions to Test Case Generators." Proc. 24th Pacific Northwest Software Quality Conference. 2006.

[4] M.B. Cohen, M.B. Dwyer and J. Shi, Interaction testing of highly-configurable systems in the presence of constraints, International Symposium on Software Testing and Analysis (ISSTA), 2007

[5] J. Bach and P. Shroeder, "Pairwise Testing – A Best Practice That Isn't." Proc. 22nd Pacific Northwest Software Quality Conference, 2004.

[6] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, "The Combinatorial Design Approach to Automatic Test Generation." IEEE Software, 13(5):83-87. 1996.

[7] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino, "Applying design of experiments to software testing." Proc. 19th International Conference on Software Engineering (ICSE '97). 1997.

[8] K. Burr and W. Young, "Combinatorial Test Techniques: Table-Based Automation, Test Generation, and Test Coverage." Proc. International Conference on Software Testing, Analysis, and Review (STAR), San Diego, CA, October, 1998.

[9] D. R. Wallace and D. R. Kuhn, "Failure Modes in Medical Device Software: an Analysis of 15 Years of Recall Data." International Journal of Reliability, Quality and Safety Engineering, 8(4):351-371. 2001.

[10] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG - a General Strategy for t-way Testing." Proc. 14th IEEE Engineering of Computer-Based Systems conference, 2007.

[11] Z. Wang, C. Nie, and B. Xu, "Generating Combinatorial Test Suite for Interaction Relationship." Proc. 4th International Workshop on Software Quality Assurance (SOQUA-2007). 2007.

[12] K. Tai and Y. Lei, "A Test Generation Strategy for Pairwise Testing." IEEE Transactions on Software Engineering, 28(1):109-111. 2002.

[13] D. Cohen, S. Dalal, M. Fredman, G. Patton, "The AETG system: An approach to testing based on combinatorial design." IEEE Transactions on Software Engineering, 23(7):437-444. 1997.

[14] K. Gundy-Burlet, J. Schumann, T. Menzies, A. Barrett, "Parametric Analysis of ANTARES Re-Entry Guidance Algorithms Using Advanced Test Generation and Data Analysis." Proc. 9th International Symposium on Artificial Intelligence, Robotics and Automation in Space. 2008.

[15] P. Williams-Hayes, "Crew Exploration Vehicle Launch Abort System Flight Test Overview." Proc. AIAA Guidance, Navigation and Control Conference and Exhibit. August 2007.

[16] T. Menzies and Y. Hu, "Data Mining for Very Busy People." IEEE Computer. November 2003.